Int J Softw Tools Technol Transfer (2010) 12:467–481
DOI 10.1007/s10009-010-0138-x

Improved usability and performance of SMT solvers for debugging specifications

David R. Cok

Published online: 19 February 2010

Abstract It is now common to construct an extended static checker or software verification system using an SMT theorem prover as the underlying logical verifier. SMT provers have improved significantly in performance over the last several years. However, their usability as a component of software checking and verification systems still has gaps. This paper describes investigations in two areas: the reporting of counterexample information and the testing of vacuity, both of which are important to realistic use of such tools for typical software development. The use of solvers in verification is more effective if the solvers support minimal unsatisfiable cores and incremental construction, evolution and querying of satisfying assignments; current solvers only partially support these capabilities.

Keywords Specification · Verification · Static checking · Vacuity checking · JML · ESC/Java · ESC/Java2 · Spec#

1 Introduction

The goal of a static program checker is to assure, without executing a program, that the program will behave as expected, where the expectations are measured by explicit and implicit specifications. Implicit specifications are provided by language rules, e.g., that forbid numerical divide-by-zero operations. Explicit specifications are provided by humans and are a statement of the constraints on input, output, and resource requirements, such as that the output of a subroutine will be a sorted array or that the screen display will accurately reflect the content of a database. Human-readable documentation is a form of specification, but one that is inaccessible to the computer. Specifications that are also computer readable can be checked automatically. There are other aspects of a programming and operating environment that also need to be correct, but in this paper we will only consider the match between a program and its specifications.

The design of verification systems for practical software has improved significantly in the past several years, encouraged by improvements in logical encoding [1,2], competitions on prover performance [3,4], and increasing coverage of programming and specification features. An additional important dimension is the way in which such tools provide information to the user. The logical information available to the user comes indirectly from a back-end prover, so the capabilities of the prover can restrict the potential of any user interface. Since most of a user's time may well be spent in correcting programs and their specifications, the amount, kind, and presentation of helpful information are important to a usable tool.

This paper describes two kinds of information that are needed to diagnose problematic programs and specifications: counterexample information and vacuity checking. Good counterexample information is needed in the case that a program does not satisfy its specifications; vacuity checking determines when a program satisfies its specifications because the specifications are trivially true. In both cases, having appropriate capabilities that are currently missing from available provers enhances both speed and quality of the information available to programmers.

The paper makes the following contributions. Section 2 provides background on the use of SMT solvers in software verification. Section 3 assesses the currently available counterexample information, describes how to translate this information such that it is more helpful to debugging specifications, and identifies changes in SMT solvers that would

D. R. Cok (✉)
Eastman Kodak Company, Rochester, NY, USA
e-mail: cok@frontiernet.net

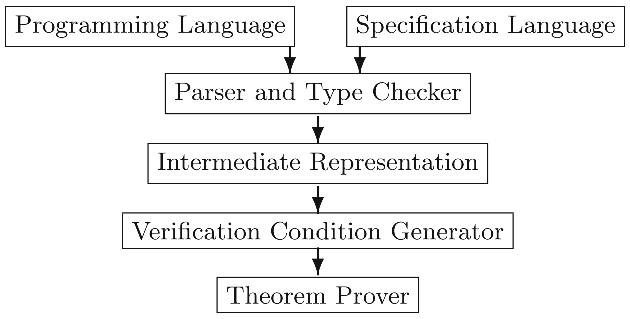

Fig. 1 The steps of software verification

further aid counterexample presentation. Section 4 presents six methods of checking for vacuous assumptions, experimentally measures their relative performance using three different SMT solvers, and presents conclusions regarding useful improvements in SMT solver functionality. Section 5 summarizes the paper's observations and describes future work.

2 Background: the design of static program checkers

2.1 Static analysis systems

Although static analyzers for program checking and software verification can be designed in many ways, a common architecture consists of the following pieces, as illustrated in Fig. 1.

- *A specification language corresponding to the target programming language*. Some specification languages are independent of programming language, such as Z [5]. However, it is now more common to have a specification language closely related to the programming language. Then, the concepts of the specification language are similar to those of the programming language, aiding learning and reducing errors. Thus, e.g., Eiffel [6] has its own built-in specification language, JML [7–9] is used for Java, the Spec# [10,11] language for C#, Anna [12] for Ada, and ACSL [13] for C. Specification languages provide syntax to express invariants (what must always be true of classes and data structure), preconditions (stating what must be true before calling a method), frame conditions (stating what may be changed by a method), and postconditions (stating what will be true on exiting a method).
- *A programming-language-dependent front-end that is responsible for parsing source code files and retrieving library contents to build a type-checked AST that represents the program and its specifications.* This is equivalent to the front-end that is used by a compiler and requires the same capabilities: scanning and parsing of source files, reading of binary files, name resolution, symbol table management, and type attribution.
- *(Optional) A program transformer that converts the AST into a language-independent intermediate representation.* Though some tools use a tool-specific representation or translate directly from an AST to verification conditions (VCs), it is increasingly convenient to use a language- and tool-independent intermediate representation. Such a representation enables reusing software components such as VC generators in multiple tools or for multiple languages. Two intermediate languages that are used for multiple programming languages are BoogiePL [14] and the Why language [15].
- *A VC generator that creates the logical statements that must be proved to assure that the program matches its specifications.* Ideally, VC generation is independent of the theorem prover to which the VCs are presented. In practice, theorem provers have different capabilities and different underlying logical systems, so that the form of the logical statements must, to some extent, be tailored to their destination. One common logical language supported by many provers is the SMT-LIB [16] language. However, this language is more suited to writing benchmarks than to interactive use during software development.
- *A theorem prover that can pronounce given logical statements as true or false.* There are by now many such provers. Some are meant as hidden batch back-end tools, e.g., Simplify [17], Yices [18,19], CVC3 [20], and Z3 [21]; others, such as PVS [18,22] and Isabelle [23], are interactive proof development environments.

Typically, these architectural components are bundled into a unified system. Some currently supported systems are Spec# for C# programs, ESC/Java2 [24] for Java, Why [15] for C or Java, and KeY [25,26] for Java.

However, the linear flow outlined above does not reflect an important requirement: a system must be effective at both identifying and explaining problems. Program or specification issues identified by the back-end prover must be communicated with as much relevant information, *expressed in the programmer's terms*, as possible. But the translation to intermediate language, to VC, to the prover's logic, and then to the results of the prover can lose the information needed in order to provide useful feedback in the source code context.

2.2 The structure of VCs

In order to see how the back-end theorem prover can aid in the program and specification debugging process, we need to understand the way in which VCs are generated. We use the basic block formulation proposed by Barnett and Leino [27].

In this procedure, a program method is transformed into an equivalent set of *basic blocks*. The basic blocks comprise the nodes of a directed acyclic graph. That is, control flows from a start block, through the graph, with branching and joining, until an end block is reached. Each block itself corresponds to a non-branching sequence of program statements. A particular execution of the program, with specific input values, will correspond to a particular path through the graph.

A basic block contains a translation of its bit of program code into a series of assumptions and assertions. The assumptions encode the various conditions that may be assumed in that block. For example, the preconditions of the method will be assumptions at the beginning of the starting block; a branch condition (or its negation) will be an assumption at the beginning of a block representing an alternative of a branch. The assertions encode conditions that must be true if the program is correct, namely, the requirements derived from the specifications.

The details of translating a program into basic blocks are described elsewhere [27] and are not repeated here. However, the next subsection contains two examples that illustrate the process. The translation includes applying program transformations to remove cycles (from loops and goto statements) and applying dynamic single assignment and passification to convert program variables into logical ones and imperative program assignments into logical conditions.

In the end, the VC for a method can be expressed as

$$(\wedge_k Q_k) \Rightarrow B_0 \tag{1}$$

where each Q_k is an equality of the form

$$B_k = ((\wedge_j A_{kj}) \Rightarrow ((\wedge_m T_{km}) \wedge (\wedge_{i \in F_k} B_i))). \tag{2}$$

The indices k and i range over all blocks; the Q_k are called block equations, the B_k are boolean block variables, the A_{kj} are assumptions, the T_{km} are assertions, F_k is the set of indices of blocks that follow block k, and index 0 corresponds to the program start block (which does not follow any other block).[1] The block equations encode the logic of the program: given the block equations, the variable for the start block (B_0) is true if and only if the program matches its specifications.

2.3 Examples

Although basic blocks and block equations are described in detail by Barnett and Leino [27], this subsection presents two examples that will assist readers in visualizing basic blocks and the resulting VCs.

[1] This actually differs slightly from the original basic block formulation in order to simplify the presentation; here, basic blocks are divided into subblocks such that assertions only occur at the end of subblocks.

```
//@ ensures \result == i > 0 ? i : -i ;
public int abs(int i) {
    if (i < 0) i = -i;
    return i;
}
```

Fig. 2 Java code demonstrating a simple conditional statement

```
B0:
    goto B1, B2;

B1:
    assume i@0 < 0 ; // assumption for then branch of if statement
    assume i@1 == -i@0; // assignment
    assume i@2 == i@1; // joining single-assignment variables
    goto B3;

B2:
    assume !(i@0 < 0); // assumption for else branch
                // no additional content
    assume i@2 == i@0; // joining single-assignment variables
    goto B3;

B3:
    assume $resultValue == i@2 ; // capturing the return value
    assert $resultValue == (i@0 > 0 ? i@0 : - i@0); // postcondition
```

Fig. 3 Basic blocks for the code in Fig. 2

```
Q0: B0 == B1 & B2

Q1: B1 == ((   (i@0 < 0)
             & (i@1 == -i@0)
             & (i@2 == i@1)
           )
           ==> B3
          )

Q2: B2 == ((  !(i@0 < 0)
             & (i@2 == i@0)
           )
           ==> B3
          )

Q3: B3 == (( $resultValue == i@2 )
           ==>
           ($resultValue == (i@0 > 0 ? i@0 : - i@0))
          )
```

Fig. 4 Block equations corresponding to Figs. 2 and 3

The first example, Figs. 2, 3 and 4, shows the blocks and block equations resulting from a simple conditional statement. There are some things to note.

- Program variables (e.g., i) are converted to logical variables ($i@0$, $i@1$, $i@2$). In single-assignment form, a new logical variable is used each time the program variable is assigned a new value.
- When control flows join, such as at the end of an if-statement, a new logical variable is used to hold the value of a program variable that may have different assignments in the different branches of the program. (In some cases, such as here, an existing logical variable could be reused for this purpose.)
- A special variable is used for the return value.
- Assignments become assumptions about the value of a new logical variable; the postcondition is represented as an assertion.

```
//@ requires j >=0;
//@ ensures \result == 0; // (j*j - j)/2;
public int triangle(int j) {
    int sum = 0;
    int i = 0;
    /*@ loop_invariant i>=0 && i<=j &&
          sum == (\sum int k; 0<=k && k<i; k);
     */
    while (i < j) {
        sum = sum + i;
        i = i + 1;
    }
    return sum;
}
```

Fig. 5 Java code demonstrating a loop, with a bug

```
B0:
    assume j@0 >= 0;
    assume sum@0 == 0;
    assume i@0 == 0;
    goto B1, B2;

B1:
    // initial check of loop invariant
    assert i@0 >= 0 && i@0 <= j@0 &&
           sum@0 == (\sum int k; 0<=k && k <i@0; k);

B2:
    // havoc sum, i - they are changed in the loop
    // check of loop invariant prior at beginning of loop body
    assume i@1 >= 0 && i@1 <= j@0 &&
           sum@1 == (\sum int k; 0<=k && k <i@1; k);
    goto B3, B4;

B3: // execute loop body
    assume i@1 < j@0 ; // while condition
    assume sum@2 == sum@1 + i@1;
    assume i@2 == i@1 + 1;
    assert i@2 >= 0 && i@2 <= j@0 &&
           sum@2 == (\sum int k; 0<=k && k <i@2; k);

B4: // loop condition fails
    assume !(i@1 < j@0); // while condition fails
    assume $resultValue == sum@1;
    assert $resultValue == 0;
```

Fig. 6 Basic blocks for the code in Fig. 5

The VC, here, is $(Q_0 \wedge Q_1 \wedge Q_2 \wedge Q_3) \Rightarrow B_0$. The reader may enjoy proving that this is true (even though the postcondition uses $>$ instead of $\geq$). Note that many assignments of values to the logical variables, such as $i@0 = 10, i@1 = 11, i@2 = 12$, satisfy the VC because the assumptions are false. It is only assignments that make all the relevant assumptions true that are critical—they may or may not satisfy the assertions as well.

A second example, in Figs. 5, 6 and 7, shows how programs with loops become acyclic. At the point where the control flow from the end of the loop joins the control flow from statements preceding the loop, the loop variable and variables that are changed in the loop body may have a variety of values. In fact, the basic block transformation allows them to have arbitrary values, constrained only by the loop invariant. This is implemented by introducing new logical variables ($i@1$ and $sum@1$) for those program variables. The only assumption about the values of these new logical variables is the loop invariant. The loop invariant is checked prior to entering the loop, and the logical translation of the

```
Q0: B0 == (( j@0 >= 0 &
             sum@0 == 0 &
             i@0 == 0
           )
           ==>
           ( B1 && B2)
          )

Q1: B1 == (i@0 >= 0 & i@0 <= j@0 &
           sum@0 == (\sum int k; 0<=k & k <i@0; k))

Q2: B2 == ((i@1 >= 0 & i@1 <= j@0 &
            sum@1 == (\sum int k; 0<=k & k <i@1; k)
           )
           ==>
           ( B3 && B4)
          )

Q3: B3 == ((i@1 < j@0 &
            sum@2 == sum@1 + i@1 &
            i@2 == i@1 + 1
           )
           ==>
           (i@2 >= 0 & i@2 <= j@0 &
            sum@2 == (\sum int k; 0<=k & k <i@2; k)
           )
          )

Q4: B4 == ((!(i@1 < j@0) &
            $resultValue == sum@1
           )
           ==>
           $resultValue == 0
          )
```

Fig. 7 Block equations corresponding to Figs. 5 and 6

loop body checks that if the loop invariant holds at the beginning of a loop, it also holds at the end. No blocks follow the loop body in the basic block formulation. It is incumbent on the loop invariant to accurately state the consequences of the loop body and to constrain the values of the loop index, so that appropriate conclusions can be drawn about the effect of the loop as a whole. (For brevity, we omit discussion of a loop variant assertion that is used to verify termination.)

This example has a bug. The postcondition incorrectly states that the result should be 0. This is in fact correct for $j <= 1$. But the assignment

$$j@0 = 2, sum@0 = 0, i@0 = 0,$$
$$i@1 = 2, sum@1 = 1, \$resultValue = 1$$

is a counterexample for the VC. Block variables B_4, B_2, and B_0 are false.

Alternately, if the sum within the loop invariant is (`\sum int k; 0 <= k && k <= i; k`), the loop invariant is initially true, but would not hold after the first iteration through the loop; a counterexample would then be

$$j@0 = 2, sum@0 = 0, i@0 = 0, i@1 = 0,$$
$$sum@1 = 0, i@2 = 1, sum@2 = 0.$$

Block variables B_3, B_2, and B_0 are false.

3 Counterexample information

3.1 The anatomy of a failed verification

If a program does not meet its stated specifications, then the VC for the program will not be true. The implication in (1) is false only when the block equations hold and B_0 is false for some assignment of values to the logical variables. Any block variable is false only when all of the assumptions of the block (the A_{kj} for that B_k) are true, but either the B_z for some succeeding block is false or some assertion T_{km} for that block is false. By induction, we can see that the VC will be false only when there is an assignment of values to variables that makes a sequence (beginning with B_0) of assumptions true but the following assertion false.

It is the task of the back-end prover, given a VC (and associated supporting axioms), to pronounce the VC valid or invalid. This is typically performed by seeking a *satisfying assignment* for the negation of the VC. If such an assignment is found, the VC is invalid, and the assignment serves as a counterexample to the validity of the program.

3.2 Presenting counterexample information

To be helpful to the user in correcting the program or the specifications, the counterexample information should be available in some appropriate form. Current systems, however, present this information in the context of the back-end prover, generally as a partial dump of the logical context of the prover. For example, the ESC/Java [28] and ESC/Java2 [24] tools use Simplify [17] as the back-end prover. Counterexample information is presented as a list of internal logical variables and their values; Fig. 8 shows a sample Java program with JML annotations and the resulting counterexample output; Figs. 9 and 10 show parts of the corresponding output from the Yices and CVC3 tools. In each tool, there is an attempt to limit the information to "relevant" variables by some heuristic. Also, logical variables are added to the VC to encode the control flow within the satisfying assignment. Thus, some tools are able to state the branch taken at each branch point, given the satisfying assignment. Nevertheless, the variables are simply listed with their values and the variable names are the encoded names of the logical variables used in the prover, including many that relate to the internal logical encoding. It takes knowledge of how the prover functions, how the VC is structured, and how the logical variable names are formed from the program variables in order to interpret the counterexample. Even then, the process of understanding the counterexample sufficiently to be able to begin debugging the specification is tedious and difficult.

```
public class ESC {
  public int[] a;
  //@ invariant a != null && a.length > 10;

  public void m(int k) {
    int kk = smallest();
    if (k > a.length) {
      k = 0;
    } else if (k < 0) {
      k = 0;
    }
    a[k] = a[kk];
  }

  /*@ ensures 0 <= \result && \result < a.length;
    @ ensures (\forall int i; 0<=i && i < a.length;
    @        a[\result] <= a[i]); */
  abstract public int smallest();
}
```

```
ESC: m(int) ...
------------------------------------------------------------------
ESC.java:12: Warning: Array index possibly too large (IndexTooBig)
    a[k] = a[kk];
     ^
Execution trace information:
    Executed else branch in "ESC.java", line 9, col 11.
    Executed else branch in "ESC.java", line 9, col 11.

Counterexample context:
    (k:5.20<2> <= intLast)
    (RES-6.13:6.13 < k:5.20<2>)
    (eClosedTime(elems) < alloc)
    (0 <= RES-6.13:6.13)
    (null <= max(LS))
    (longFirst < intFirst)
    (fClosedTime(a@pre:2.15) < alloc)
    (vAllocTime(this) < alloc)
    (intLast < longLast)
    (1000001 <= intLast)
    ((intFirst + 1000001) <= 0)
    (11 <= k:5.20<2>)
    typeof(this) <: T_ESC
    elemtype(T_int[]) == T_int
    T_int[] <: T_int[]
    null.LS == @true
    typeof(this) <: T_ESC
    typeof(state-6.13:6.13) <: T_anytype
    (null <= max(LS))
    typeof(RES-6.13:6.13) <: T_int
    typeof(k:5.20<2>) <: T_int
    T_int[] <: arrayType
    typeof(tmp0!a:12.4) == T_int[]
    typeof(tmp0!a:12.4) <: T_int[]
    T_int <: T_int
    this.(a@pre:2.15) == tmp0!a:12.4
    arrayLength(tmp0!a:12.4) == k:5.20
    a@pre:2.15 == a:2.15
    k:5.20<2> == k:5.20
    T_bigint == T_long
    EC-6.13:6.13 == ecReturn
    after@6.13-6.13 == state<1>
    elems@pre == elems
    state@pre == state
    k:5.20<1> == k:5.20
    alloc@pre == alloc
    !typeof(tmp0!a:12.4) <: T_void
    typeof(this) != T_void
    !T_java.lang.Object <: T_java.io.Serializable
    !typeof(this) <: T_void
    T_int[] != T_void
    bool$false != @true
    this != null
    tmp0!a:12.4 != null
    EC-6.13:6.13 != ecThrow

------------------------------------------------------------------
```

Fig. 8 Java example with JML specifications and counterexample information from ESC/Java2

The Spec# system builds on this same philosophy, but embeds the user interaction in an IDE, an improvement in user-friendliness. The trace information is shown overlaid on the program itself within the source code editor, and the few

```
(= $assumption25 true)
(= $assumption26 true)
(= $assumption27 true)
(= assumeCheck$828$Precondition false)
(= _$BL$bodyBegin true)
(= $assumption29 false)
(= assert$828$828$Precondition$30 true)
(= $assumption33 true)
(= $assumption35 false)
(= _$BL$828$afterCall$36 true)
(= _$BL$828$afterCallExc$37 true)
(= $assumption38 false)
(= _$BL$return true)
(= _$BL$exception true)
(= $assumption39 false)
(= $assumption40 false)
(= length length$0)
(= T$java.lang.Object$$A 1)
(= _terminationVar$$$0$1 830)
(= _terminationVar$$$$828$34 1)
...
```

Fig. 9 Sample counterexample information from Yices

```
ASSERT (NULL = NULL);
ASSERT (distinct$ = (LAMBDA (uf_0: TYPE$):
   IF (uf_0 = T$tt_TestJava) THEN 101
   ELSE IF (uf_0 = T$java_lang_Object) THEN 102
   ELSE IF (uf_0 = T$java_lang_Throwable) THEN 103
   ELSE 104 ENDIF ENDIF ENDIF));
ASSERT (subtype$ = (LAMBDA (uf_1: TYPE$, uf_2: TYPE$): TRUE));
ASSERT (_alloc$$ = (LAMBDA (uf_3: REF): 101));
ASSERT (typeof$ = (LAMBDA (uf_4: REF):
   typeof$(__exception_0_20_11)));
ASSERT NOT __BL_Start;
ASSERT _assumption4;
ASSERT _assumption5;
ASSERT _assumption6;
ASSERT _assumption7;
ASSERT _assumption8;
ASSERT _assumption9;
ASSERT _assumption10;
ASSERT NOT assumeCheck_20_Precondition;
ASSERT __BL_bodyBegin;
...
```

Fig. 10 Sample counterexample information from CVC3

variables whose values are selected to be shown are translated back into program variable names. However, an insufficient set of values is selected from the counterexample, and the user is not able to explore the counterexample in any way.

Other systems (e.g., Why or KeY) provide a view of the proof process from the point of view of a formal logic expert. There is an explicit view into the VCs generated by a program; there are means to attempt a variety of provers on specific subgoals, and means to inspect which subgoals are proved, which are not, and so on. These tools target a user familiar with the details of the formation and proof of VCs.

None of these systems provide means to query the counterexample in the context with which the programmer is most familiar—the software code itself and its control flow. A typical bug is subtle and requires investigating program values and their ramifications at various points in the program. The counterexample information is crucially valuable, but it can be used effectively only if presented more accessibly.

3.3 Improved presentation of counterexample information

We can improve the understanding of the counterexample information by presenting it in the context of a program's source code and control flow. In particular, we can trace through the program, statement by statement, according to the values of variables in the counterexample, reaching the false assertion. The counterexample provides enough information to retrace the control flow and to give the values of variables and subexpressions along the way. Specific provers return differing amounts of information, but appropriate queries to the prover, if supported, can supply the values of any desired program variables and subexpressions. Presenting this information in association with the source code and with the details obtained by this pseudo-execution provides the user with a clear picture of the erroneous state of the program, rather than the flat view of a subset of the logical variables provided currently.

A control-flow-oriented presentation is used in other contexts. Tools that use symbolic execution have natural access to the control path that leads to a problematic state, and presentations of that control path are used even in commercial tools [29]. Similarly, tools (e.g., the JACK tool [30]) that give the user visibility to the proof obligations that the user may need to interactively prove can show the statements of a program that contribute to the proof obligation. Here, the contribution is to provide a similar capability for counterexample information.

First, we note that the basic block formulation contains within it the needed control flow information. A satisfying assignment that serves as a counterexample will have B_0 false and will correspond to a path through the acyclic graph that ends with the assertion that is false under this assignment. The block variable B_k for each block on this path will be false. The relevant execution path can be found by tracing through the directed acyclic graph of blocks, either forward from the start block or in reverse from the block containing the false assertion, to find the appropriate sequence of blocks with false block variables in the satisfying assignment. Note that the values of logical and block variables that are not part of this chain of blocks may have arbitrary assignments. Hence, it is not sufficient to search for any block with a false block variable—only those that form a path from start to false assertion are relevant.

A small, constructed example of this procedure is shown in Fig. 11. The assignment $i@0 = 0$, $j@0 = 0$ is a counterexample for the VC (a satisfying assignment for the negation of the VC); in particular, it sets B_0, B_1, and B_3 false, and B_2, B_4, and B_5 true. Both assertions are false, but the relevant control flow path is from block B0 to B1 to the assertion in block B1. Since block B3 follows from block B2 and the block variable B_2 is true, it is irrelevant to the counterexample that B_3 is false. Sometimes a false block such as B3 is

```
------------ Sample program ------------

public void sample(int i, int j) {
    if (i <= 0) {
        assert i < 0;
    } else {
        if (j <= 0) {
            assert j < 0;
        }
    }
}

------------ Basic Blocks --------------

B0: goto B1, B2;

B1: assume i@0 <= 0;
    assert i@0 < 0;
    goto B5;

B2: assume !(i@0 <= 0);
    goto B3, B4;

B3: assume (j@0 <= 0);
    assert j@0 < 0;
    goto B5;

B4: assume !(j@0 <= 0);
    goto B5;

B5: // end

------------ Verification Condition --------------

VC: (Q0 && Q1 && Q2 && Q3 && Q4 && Q5) ==> B0;

Q0: B0 == (B1 && B2);

Q1: B1 == ((i@0 <= 0) ==> ((i@0 < 0) && B5));

Q2: B2 == ((!(i@0 <= 0)) ==> (B3 && B4));

Q3: B3 == ((j@0 <= 0) ==> ((j@0 < 0) && B5));

Q4: B4 == ((!(j@0 <= 0)) ==> B5);

Q5: B5 == true;
```

Fig. 11 Java code demonstrating a simple conditional statement

infeasible (never executed), in which case the fact that it is false is misleading; other times (such as in this case) there may be other counterexamples (e.g., $i@0 = 1$, $j@0 = 0$) in which B3 is feasible and that identify other program or specification errors.

Second, we need to retrieve the values of expressions and assignments for each program variable at arbitrary points along the control flow path. Those values are readily available for program variables corresponding to specific logical variables that are included in the counterexample information. However, sometimes one wants to query the value of a subexpression, such as the index of an array element assignment or the return value of a function call for which no specific logical variable is defined or returned by the prover. For these subexpressions, we can proceed as follows:

- To obtain the value of a desired expression *expr*, assert to the prover the negated VC and the additional equation X = *expr*, where X is a new variable name. Multiple expressions can be queried using an appropriate conjunction.
- If the original negated VC was satisfiable, the logical context will still be satisfiable and the value of X will be the desired value of the subexpression.

However, resending to the prover the entire negated VC each time we wish to query the value of a subexpression is inefficient. We prefer to query the existing state of the prover after it has generated a satisfying assignment. Most current provers do allow incremental addition of new constraints to an existing logical context. In such an incremental mode, one would assert the negated VC to the prover and check for satisfiability; then one would assert the new equations, recheck for satisfiability and inspect the new counterexample for the values of the desired expressions. However, there is no guarantee that the recheck will find the same satisfying assignment, even though the original assignment will still be satisfying. We can simulate the situation that the satisfying assignment stays the same (augmented with the new variables) by asserting the entire counterexample back to the prover, along with the new queries. This still gives a satisfying assignment but results in one that has the same values on all variables that have already been reported.

With this ability to query a counterexample for the values of arbitrary subexpressions, a text-based interface or a GUI can provide the user the ability to explore the implications of a counterexample, in the context of the program text, rather than the user simply receiving a list of the values of mostly irrelevant variables.

This capability allows reverse debugging as well. A static, logic-based approach as described above naturally retains a record of state changes and provides symbolic information about any previous state. Within generally available runtime debuggers, one can usually set breakpoints and inspect the current state of a stopped program. However, one typically cannot look back to previous program states. (There are research debuggers that have provided reverse stepping by taking snapshots of the state or logging state changes; e.g., GDB intends to add support for reverse debugging in 2009, and Cook [31] and later ODB [32] implement this feature for Java.)

In order to efficiently execute the above procedure, a backend solver needs to have the following capabilities:

- the ability to add additional constraints to a current logical context, checking for satisfiability as new groups of assertions are added, extending the current satisfying assignment incrementally, if possible;
- the ability to query a satisfying assignment for the implied value of variables or expressions, without the current assignment changing (although it may perhaps be augmented).

The first item above is fairly standard in SMT solvers, though rechecking for satisfiability may cause a completely

Fig. 12 An example of improved counterexample information from OpenJML. The @L notation indicates a line number. A remaining confusing aspect is Yices mapping of user-defined types (e.g., object references) to integers

```
Checking method ESC.m(int)
ESC.java:12: warning: The prover cannot establish an assertion (PossiblyTooLargeIndex) in method m
    a[k] = a[kk];
     ^

Trace
ESC.java:5:     Parameter     k  = 11
ESC.java:3:     Invariant this.a != null && this.a.length > 10
                                4    = this
                                5    = this.a
                                true = this.a != null
                                4    = this
                                5    = this.a
                                11   = this.a.length
                                true = this.a.length > 10
                                true = this.a != null && this.a.length > 10
ESC.java:5:     Precondition true
ESC.java:6:     ReceiverEvaluation this@L6 = this
ESC.java:6:     Called method returned normally, return value = 10 [$$result@L6]
ESC.java:6:     Postcondition (normalTermination@L6 ==> (((\forall int i;
    0 <= i && i < this@L6.a.length;this@L6.a[$$result@L6] <= this@L6.a[i]))))
ESC.java:3:     Invariant this@L6.a != null && this@L6.a.length > 10
ESC.java:6:     Assignment kk = 10  [$$result@L6]
ESC.java:7:     BranchCondition = false  [(k > this.a.length)]
ESC.java:9:     BranchCondition = false  [(k < 0)]
ESC.java:9:     DSA       k@L11 = k
ESC.java:12:    Assert [PossiblyTooLargeIndex] false   [(true ==> k@L11 < this.a.length)]
                                11   = k@L11
                                4    = this
                                5    = this.a
                                11   = this.a.length
                                false = k@L11 < this.a.length
                                false = (true ==> k@L11 < this.a.length)
```

new check to be performed, rather than building on the results of the previous check. The second item is less common. The SMT-LIB API, currently under development, is also a place where such functionality should (and is beginning to) be present, though that standard would then need to be supported by functionality in actual tools.

A few other aspects should be mentioned that are quite straightforward to provide (although they are not in current tools).

- The assumptions and assertions that make up a basic block program are generated by a variety of source code constructs. By recording the source of specific assumptions and assertions, information returned by the prover about them can be portrayed in the context of the source code. For example, both assignments and branch conditions generate assumptions about variables, but, as in Fig. 12, with the right internal information, the presentation can be appropriately different.
- Similarly, the logical variables used by the prover are typically a cryptic encoding of various pieces of information: the variable name, its declaration location, and the location of its most recent update (as you can see in Fig. 8). Without detailed knowledge, these names are inscrutable to the programmer. Translating them back into the context of the program's source code and succinctly displaying the relevant information are essential for easy understanding. (SMT-LIB [33] allows for annotations of this sort in its concrete syntax, but this capability has not yet been integrated into verification systems.)
- In this paper, counterexample information is shown in textual form. Within a visual source-code editor, the appropriate parts of the control flow can be suitably highlighted.
- Finally, though not discussed here, a tool can perform additional analysis (e.g., dependency analysis) of the counterexample information and the content of the program's assumptions and assertions in order to identify more precisely the aspects of a program that lead to the counterexample.

These techniques were implemented for experimentation in the OpenJML tool, a JML and ESC implementation built with OpenJDK [34] and Yices as the prover. Figure 12 shows the generated counterexample information for the same code as in Fig. 8. Subexpression information is shown for just a few expressions for space reasons; also, column information is omitted. With this presentation, it is more evident that the problem with the code occurs when the argument of `m(int k)` has the same value as the length of the array `a`.

4 Vacuity checking

Vacuity checking is a second area in which improved prover capability is needed in order to support a better user experience. Vacuity checking verifies that if a prover pronounces that a program meets its specifications, it is not because the specifications are trivially satisfied. If the preconditions of a method are inconsistent (equivalent to `false`), then any subsequent assertion will be deemed valid by the theorem prover, i.e., will be vacuously true. Similarly, if an inadvertently false assumption is included in the text of a program, any subsequent assertions will be deemed valid.

Vacuous assumptions are, of course, human errors. But without additional checks, a program with vacuous assumptions is reported to be just as valid as one that correctly meets its specifications. This is a well-known problem in model checking, e.g. [35,36]; there is also some initial work in the context of static analysis [37]. In requirements modeling, this task is also known as consistency or satisfiability checking, e.g. [38,39]. In the case of a formal requirements model, one can check for global consistency by finding an instance that satisfies all the constraints of the model. The fundamental logical task is similar to that needed for software verification, but the formulation of the satisfiability task is different in the two applications.

Note first that if a VC for a module is invalid, there is no need to do vacuity checking. Rather, vacuity checking is called for when the VC is reported as valid. Second, a module may have many assumptions. Most methods of checking for vacuous assumptions require that each assumption be checked individually (or at least each assumption sequence must be tested individually). Hence, multiple vacuity checking tests would be run for a given program module and the performance of the checker is important. The following sections describe a number of ways to check for vacuous assumptions and compare their performance.

4.1 Methods of vacuity checking

4.1.1 A-InsertAssert. The assert false technique

The traditional test that is performed in program checking systems is to insert an appropriately placed **assert false** statement within the program and then recheck the program. The inserted assertion is placed after the assumption in question, or perhaps after a sequence of assumptions within a basic block. If the **assert false** triggers an assertion violation, then any assumptions prior to that assertion are at least not vacuous; however, if no assertion violation occurs despite the explicit **assert false**, then that assertion is never logically reached, and there is likely a problem.

Though testing each assumption individually gives the most precision, one can begin by testing each basic block individually. In this case, one inserts an **assert false** statement at the end of each block in turn. If the assertion causes the VC to be invalid, then the assumptions in that block are not vacuous; if the VC is still valid, then the assumptions in the block need to be tested individually.

This method of inserting test assertions has been a manual idiom in ESC/Java2 and is applied automatically by some tools, e.g. [37]. It requires a separate reformulation of the VC and a separate test of satisfiability for each assumption or block being checked.

4.1.2 B-TruncAssert. The truncated assert false technique

Method A-InsertAssert of the previous subsection can be improved by noting the following: none of the statements following a false assertion are needed. If the assertion is executed, execution will stop because it is false, and nothing after it is ever executed. If the assertion is not executed (and the VC is reported as still valid), then nothing after it is executed either. Hence, we can shorten the program by eliminating any statements within the basic block after the point of inserting the false assertion; we can also ignore any other basic blocks that follow only the block containing the inserted assertion. This is the technique used by Janota et al. [37] in their reachability analysis.

In this way, the VC generated from the module augmented with the inserted assertion is simpler than the original VC and may be more quickly tested. In particular, the preconditions of a method are translated as assumptions that begin a VC. Those preconditions must also be tested for vacuity; when a false assertion is placed after them, the entire body of the method can then be ignored.

4.1.3 C-PushPop. The push-pop technique

Techniques A-InsertAssert and B-TruncAssert require repeated testing of similar VCs. For example, each VC will contain a large background predicate that states many definitions and theorems about the logical behavior of the operations and constructs of the programming language. Logical statements encoding the properties of classes referenced by the module under test also need to be generated and presented to the theorem prover. It seems inefficient to repeatedly generate and test VCs with largely the same content.

Most SMT solvers have a checkpointing capability. That is, the state of the logical context can be saved ("pushed"), further operations performed, and then the saved state can be restored ("popped"). This can be used for vacuity testing as follows:

- Choose a new logical variable name, e.g., *assumptionNumber*.
- Assign a unique positive integer to each assumption to be tested.
- Insert after each assumption to be tested the assertion *assert assumptionNumber != M*, where *M* is the number of that assumption.
- Generate the resulting VC; present it to the solver.
- Then repeatedly, for N equal to 0 and for N equal to each of the integers assigned to assumptions, do the following:

- save the state
- add the assumption *assumptionNumber* = *N* to the solver's logical state
- test the resulting VC for validity
- restore the state.

Now,

(a) When the equality *assert assumptionNumber* = *0* is added to the logical state and tested, the VC is equivalent to the VC of the original program. With this assignment of 0 to the variable *assumptionNumber*, all the inserted assertions are true, and the truth of the original VC is unchanged.
(b) When the equality *assumptionNumber* = *M* is added to the logical state and tested, the VC is equivalent to the original VC with a single *assert false* after the assumption numbered *M*.

Consequently, this procedure performs the same tests as Technique A-InsertAssert, but without needing to restart the SMT solver for each test. Note that the optimization of testing each basic block rather than each assumption can be applied here, but the optimization of Technique B-TruncAssert cannot.

4.1.4 D-Retract. The retraction technique

Another capability of some solvers is to be able to retract specific logical statements presented to the solver. With this technique, we do not need to repeatedly save and restore the entire logical state. Rather, we proceed as follows:

- Choose a new logical variable name, e.g., *assumptionNumber*.
- Assign a unique positive integer to each assumption to be tested.
- Insert after each assumption to be tested the assertion *assert assumptionNumber != M*, where *M* is the number of that assumption.
- Generate the resulting VC; present it to the solver.
- Then repeatedly, for N equal to 0 and for N equal to each of the integers assigned to assumptions, do the following:
  - add the logical statement *assumptionNumber* = *N* to the solver's logical state
  - test the resulting VC for validity
  - retract the statement just made.

This also replicates all of the tests of individual assumptions performed in Technique A-InsertAssert. As for Technique C-PushPop, the optimization of testing each basic block rather than each assumption can be applied here, but the B-TruncAssert optimization cannot.

4.1.5 E-Search. The search technique

The techniques so far described test each assumption by inserting a false assertion, one at a time. It is possible to combine some of these tests and have the solver decide which ones might be false. Consider the following technique:

- Choose a new logical variable name, e.g., *assumptionNumber*.
- Assign a unique positive integer to each assumption to be tested.
- Insert after each assumption to be tested the assertion *assert assumptionNumber != M*, where *M* is the number of that assumption.
- Generate the resulting VC; present it to the solver.
- Then repeatedly, do the following:
  - test to see if the VC is valid
  - if the VC is valid, then *all* remaining assumptions are vacuous (and the iteration ends)
  - if the VC is invalid, the counterexample demonstrating invalidity will have an assignment for the variable *assumptionNumber*. The value of *assumptionNumber* will correspond to one of the assumptions. Since the assertion corresponding to that assumption is false, that assumption is not vacuous.
  - add the logical statement *assumptionNumber != N* to the logical context and repeat.

This procedure allows the solver to choose which of the inserted assertions is false. The most common case is that all of the assumptions under test are non-vacuous; in that case, the total number of prover invocations is the same as in any of the previous techniques. If there are multiple vacuous assumptions, the total number of tests will be reduced.

4.1.6 F-UnsatCore. The unsatisfiable core technique

When a negated VC is unsatisfiable, some provers (e.g., Yices) can also report a subset of given assertions that, by themselves, are unsatisfiable. This set is called an unsatisfiable core. If an assertion to the prover is not part of the unsatisfiable core, then it is irrelevant to the proof of validity. So suppose we add, after an assumption to be checked, the program assertion `assert X`; in addition, we separately assert to the prover a statement equivalent to `X = true` (`X` being a previously unused variable name). Since `X` is true, the VC is logically equivalent to the original VC, so we can build in this assertion when the VC is first constructed. However, if the assertion `X = true` turns out to be irrelevant (i.e., it is not part of the unsatisfiable core), then it does not matter whether X is true or false, and there must be something vacuous prior to the assertion. This allows a test for vacuity with only the overhead of generating the unsatisfiable core and without needing to do additional checks for satisfiability.

There is a hitch, however: the core may not be minimal. If the core is not minimal, then there may still be irrelevant assertions in the given core. Consequently, anything left in the core, including any non-vacuous assumption (which, one hopes, is all of them), still needs to be tested individually. Accordingly, it would be particularly advantageous to know that the reported core is minimal.

Thus, the quality of information available (efficiently) to the user can be enhanced if SMT solvers have these capabilities:

- When reporting that a logical context is unsatisfiable, a prover also reports an unsatisfiable core;
- The prover also reports that the core is minimal, when it can do so efficiently.

4.2 Performance of vacuity testing techniques

In typical use, vacuous assumptions will be inadvertently introduced into a specification, detected by an appropriate tool, and then corrected. A typical program module will contain many implicit or explicit assumptions. Most of the time, nearly all assumptions will be non-vacuous. Hence, the efficiency of checking vacuity is important to a productive software development workflow.

The performance of the techniques described in the previous section was measured as follows:

- The OpenJML tool was used to translate sample Java programs into VCs with testable assumptions.
- Three different SMT solvers were assessed: Simplify, Yices, and CVC3. However, the performance comparisons always compare different techniques using the same solver; different solvers were not compared against each other.
- Since most existing code is either not annotated with user assumptions or has already been checked for vacuity, these performance tests were performed on constructed programs that contained dead code from unused conditional and switch statement branches. The scale of the test programs was varied to provide performance checks across a range of time scales and locations of the vacuous assumptions.
- Each measurement was repeated at least three times; comparisons were made against the median baseline time (the baseline was usually the A-InsertAssert technique).

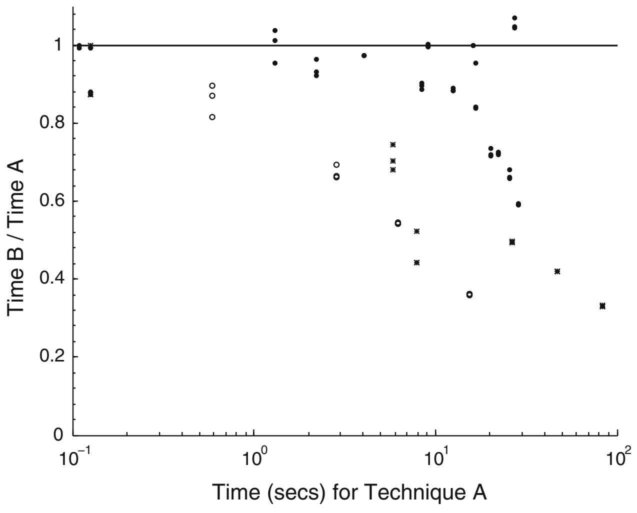

Fig. 13 Ratio of times for Technique B-TruncAssert to A-InsertAssert versus the time by Technique A-InsertAssert (*filled circles* Yices, *open circles* Simplify, and *asterisks* CVC3 provers)

These test scenarios are atypical of common programming environments in two ways.

- The test programs are simpler than most programs would typically be, with less dependence on other modules. I expect that using more complex programs would accentuate the value of the optimizations described in the sequel.
- The test programs contain assumptions mostly from branch and loop statements; typical vacuous assumptions are also (perhaps more) likely to arise from incorrect axioms and invariants.

4.2.1 Technique B-TruncAssert versus A-InsertAssert

Figure 13 shows the ratio of times using the B-TruncAssert technique to A-InsertAssert, as a function of the time using Technique A-InsertAssert. Recall that the difference between these two techniques is that in Technique B-TruncAssert, the VC is truncated after known false assertions. The comparison shows that, particularly for longer running tests, the truncation can save up to 60% of the running time. The savings is greater for Simplify and CVC3 than for Yices, prompting the conjecture that Yices may implement this optimization internally.

4.2.2 Technique C-PushPop versus A-InsertAssert

Figure 14 shows the ratio of times using the C-PushPop technique to A-InsertAssert, as a function of the time using Technique A-InsertAssert. This comparison shows a clear improvement across all tests and provers of 30–90%. The time saved using stored state is an improvement over the truncation technique (Technique B-TruncAssert).

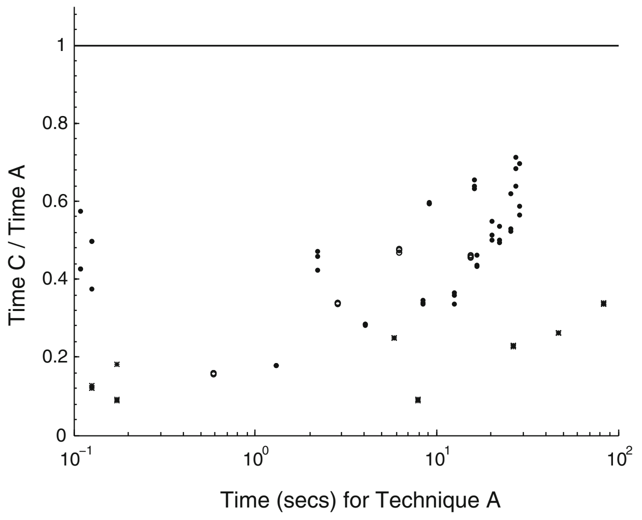

Fig. 14 Ratio of times for Technique C-PushPop to A-InsertAssert versus the time by Technique A-InsertAssert (*filled circles* Yices, *open circles* Simplify, and *asterisks* CVC3 provers)

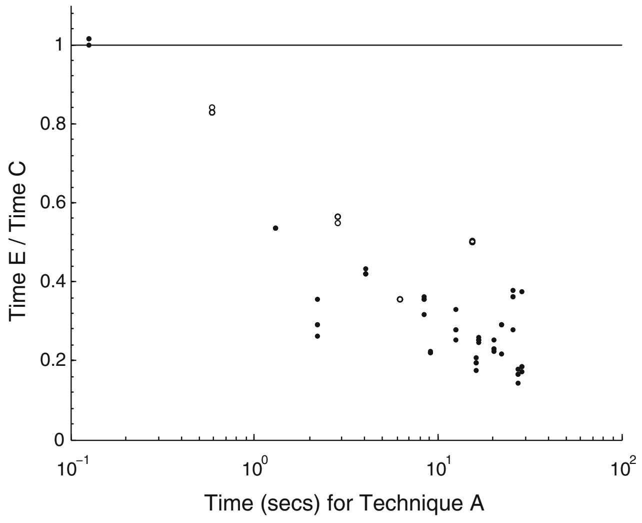

Fig. 16 Ratio of times for Technique E-Search to C-PushPop versus the time by Technique A-InsertAssert (*filled circles* Yices and *open circles* Simplify provers)

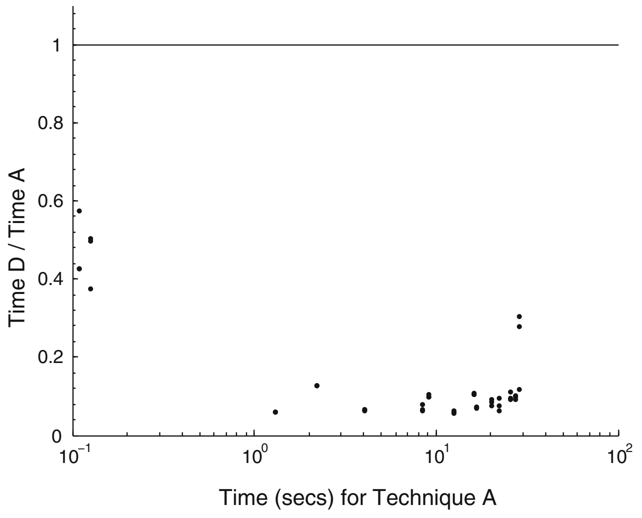

Fig. 15 Ratio of times for Technique D-Retract to A-InsertAssert versus the time by Technique A-InsertAssert (Yices prover only)

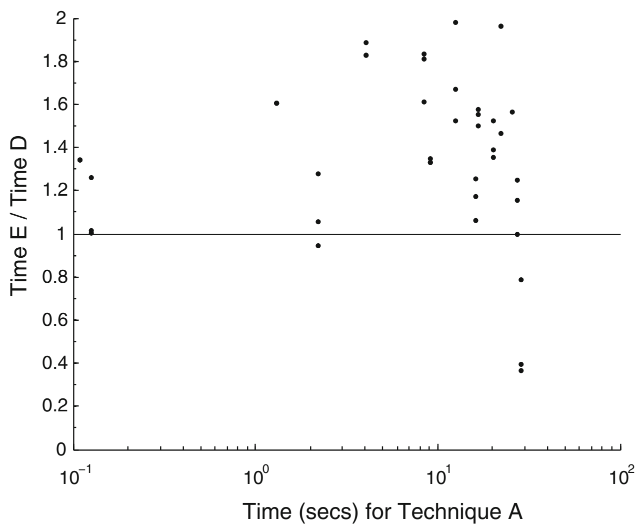

Fig. 17 Ratio of times for Technique E-Search to D-Retract versus the time by Technique A-InsertAssert (Yices prover only)

4.2.3 Technique D-Retract versus A-InsertAssert

The retraction technique is only available in the Yices prover. Figure 15 shows the performance of this technique compared to the baseline. Most test scenarios show a further improvement over Technique C-PushPop, with times that are about 10% of the baseline. Very short tests show less improvement; presumably those times are dominated by initialization.

4.2.4 Technique E-Search versus C-PushPop and D-Retract

Technique E-Search also shows substantial improvement over the baseline (Technique A-InsertAssert), so it is more interesting to compare E-Search to the best techniques so far: C-PushPop and D-Retract. Technique E-Search could be implemented only in Yices and Simplify; technique D-Retract is only available for Yices. Figure 16 shows the ratio of times using Technique E-Search vs. Technique C-PushPop for the various tests: Fig. 17 shows the comparison between E-Search and D-Retract. Though E-Search does generally better than C-PushPop, it usually is worse than D-Retract. In fact, the comparison depends both on the prover and on the details of the particular test.

In Technique D-Retract, we test each assumption individually with an explicit false assertion. In Technique E-Search, we allow the prover to find an assertion which, if false, will invalidate the VC. This more complicated problem makes

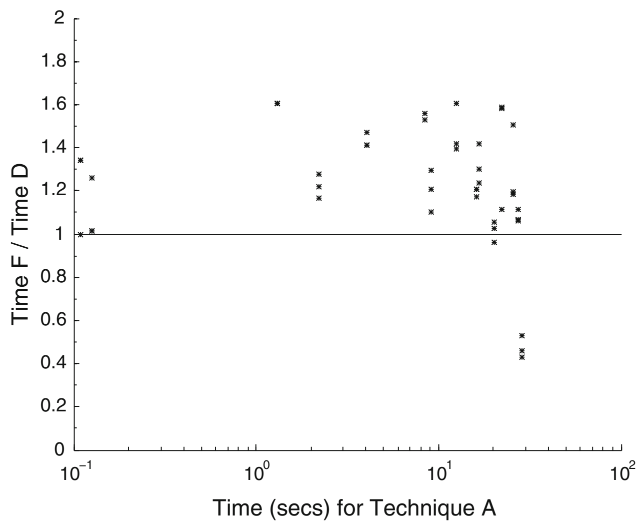

Fig. 18 Ratio of times for Technique F-UnsatCore to D-Retract versus the time by Technique A-InsertAssert (Yices prover only)

each prover invocation of E-Search take longer than each prover invocation of D-Retract.

We do expect Technique E-Search to require fewer prover invocations. However, the prover invocations that are saved are only those for which there are vacuous assumptions. If there are only a few vacuous assumptions, then E-Search will require nearly as many prover invocations as D-Retract, and therefore perform worse.

This observation parallels one made by Leino et al. [40]. Those authors found that individually testing a number of logical assertions was faster than a single test of the conjunction of the assertions. They conjectured that the search techniques or quantifier heuristics are sometimes less efficient when confronted with a larger, more complex logical assertion to test. Similarly, in the situation presented here, the more complicated VC takes longer to check than does an equivalent but larger set of simpler assertions.

4.2.5 Technique F-UnsatCore versus D-Retract

The unsatisfiable core technique is available only in the Yices prover. Figure 18 shows the comparison to D-Retract, the best other technique for Yices. In the context of these tests, technique F-UnsatCore performs worse than D-Retract. In the Yices prover, in order to obtain information about the unsatisfiable core, one must turn on the "evidence" option. This degrades the performance of each validity test. Furthermore, the results showed that the assertions being added to test for vacuous assumptions were, nearly always, still present in the unsatisfiable core, even when the core would have been unsatisfiable without them. That is, the core was generally far from minimal and sometimes no different than the entire original assertion set. Also, in practice, we would expect there to be few vacuous assumptions compared to the number of non-vacuous assumptions. Thus, little benefit is gained for the cost of generating the unsatisfiable core. A qualitative observation is that the degree to which the reported core is minimal depends on the details of how the VC is constructed (as well as on the particular internal algorithms of the prover); equivalent VCs, expressed in different but equivalent logical forms, can result in markedly different unsatisfiable cores.

5 Concluding observations and future work

The discussion above presented improvements in the use of verification systems for debugging programs and specifications. First, we described a novel method of presenting counterexample information to the user. In this style, variable and subexpression values are presented in a trace of the program's control flow, as if the program were executed given the value assignments of a particular counterexample. This presents the counterexample information in the programming context, rather than in the theorem proving and VC context.

Secondly, we noted that the logical state manipulation features of some provers allow alternate ways to do vacuity checking. Performance comparisons show that vacuity checking using saving and restoring of logical state in order to do a number of nearly similar validity checks can save substantial amounts of time. However, the reporting of unsatisfiable cores by provers (at least, by Yices) is insufficiently minimal to provide a reliable performance improvement. Similarly, no gain is obtained by constructing the VC tests in a fashion in which the algorithm repeatedly reduces the set of assumptions being checked until only the vacuous ones remain.

Both of these innovations for user workflow benefit from support by the underlying theorem prover. Current provers generally have the ability to incrementally add additional constraints to a logical context, checking for satisfiability as new groups of assertions are added. Effective workflow would be further helped by these additional capabilities:

- the ability to query a satisfying assignment for the implied value of variables or expressions without the current assignment changing (although it may perhaps be augmented);
- the ability to add new constraints that are consistent with the current assignment, without the current assignment changing (although it may perhaps be augmented);
- for sets of logical assertions used here, Yices showed better performance using assert and retract instead of push and pop operations, suggesting that other provers might implement a corresponding technique;
- when reporting that a logical context is unsatisfiable, a prover also reports an unsatisfiable core and, when efficiently possible, that it is minimal.

These comparisons of vacuity-checking performance used programs constructed to have infeasible assumptions or branches. It would also be helpful to understand the relative performance on large-scale natural programs. However, the most useful checks are those of the consistency of user-specified assumptions, so such a study is left for future work when there are a larger number of adequately specified (but not error-free!) programs available. However, we can note first that vacuity checks are only necessary if a program or module is reported to have no bugs. Second, large portions of a program's specifications will remain unchanged as a program is developed. Thus, any system that allows incremental checks will be able to usefully avoid doing vacuity checks on every change; once a set of specifications for a module is developed and checked, it is reasonable to omit checking them regularly until there is reason to doubt their consistency or until a final pass over the program is required. Both of these considerations will reduce the overhead of including vacuity checking in a full verification system.

There are now a number of very capable SMT provers available for use. This paper compared three of them, finding significant differences in their response to different optimizations. There is, however, no detailed information about the internal algorithms within the various provers. Thus, one is not able to relate these differences to differences in internal heuristics. Nor is one even able to explain why a given prover responds as it does to simpler or more complex VCs. Clearly, the static analysis and software verification community would benefit from better explication of prover internals and a better understanding of the interaction between prover algorithms and the structure of logical assertions.

The investigations described here also inspire some additional avenues of study:

- The intuitive improvement of Fig. 12 over Fig. 8 should be informed by field studies of actual debugging practice and the value of various aids.
- The performance of the mechanisms described for obtaining counterexample and vacuity information should be measured on large-scale code bases.
- The counterexample information shows what value a variable has. Information from the prover explaining why it has that value would be even more useful.
- In real-world scenarios, the VCs presented to an SMT solver include a mass of axioms, most of which will be irrelevant to the particular proof. Both performance and presentation need to be assessed and improved for this situation.

The work to date in improving the back-end SMT provers has concentrated on raw speed and accuracy against benchmarks, driven by competitions. The discussion in this paper points out the need for advances on another front as well: the capabilities needed for effective support of a particular application of the provers, namely, static program checking.

References

1. Flanagan, C., Saxe, J.B.: Avoiding exponential explosion: generating compact verification conditions. SIGPLAN Not. **36**(3), 193–205 (2001)
2. Leino, K.R.M.: Efficient weakest preconditions. Inform. Process. Lett. **93**(6), 281–288 (2005)
3. Barrett, C., de Moura, L., Stump, A.: Design and results of the 1st satisfiability modulo theories competition (SMT-COMP 2005). J. Autom. Reason. **35**(4), 373–390 (2005)
4. The SMT-COMP web site provides results of the SMT competition and links to the system descriptions of the participants. http://smtcomp.org
5. Diller, A.: Z: An Introduction to Formal Methods. 2nd edn. Wiley, New York (1994)
6. Meyer, B.: Object-oriented Software Construction. Prentice Hall, New York, NY (1988)
7. Leavens, G.T., Baker, A.L., Ruby, C.: JML: a notation for detailed design. In: Kilov, H., Rumpe, B., Simmonds, I. (eds.) Behavioral Specifications of Businesses and Systems, pp. 175–188. Kluwer Academic Publishers, Boston (1999)
8. Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D.R., Müller, P., Kiniry, J., Chalin, P., Zimmerman, D.M.: JML Reference Manual. Available from http://www.jmlspecs.org (May 2008)
9. JML web site. http://www.jmlspecs.org
10. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: an overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS 2004), vol. 3362 of Lecture Notes in Computer Science, pp. 49–69. Springer (2005)
11. Spec# web site. http://research.microsoft.com/SpecSharp
12. Luckham, D.C., von Henke, F.W., Krieg-Brückner, B., Owe, O.: ANNA—A Language for Annotating Ada Programs, Reference Manual, vol. 260 of Lecture Notes in Computer Science. Springer (1987)
13. ACSL web site. http://www.frama-c.cea.fr/acsl.html
14. DeLine, R., Leino, K.R.M.: BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70. Microsoft Research (2005)
15. Why web site. http://why.lri.fr
16. The SMTLIB web site hosts benchmarks for SMT solvers and defines a common SMT input language. http://combination.cs.uiowa.edu/smtlib
17. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J ACM **52**(3), 365–473 (2005)
18. Rushby, J.: Tutorial: automated formal methods with PVS, SAL, and Yices. In: Fourth IEEE International Conference on Software Engineering and Formal Methods, 2006 (SEFM 2006), 11–15 September 2006
19. Yices web site. http://yices.csl.sri.com
20. Barrett, C., Tinelli, C.: CVC3. In: Damm, W., Hermanns, H. (eds.) Computer Aided Verification, Proceedings of 19th International Conference (CAV 2007) Berlin, Germany, 3–7 July 2007, vol. 4590 of Lecture Notes in Computer Science, pp 298–302. Springer (2007)
21. de Moura, L.M., Bjørner, N.: Z3: An Efficient SMT Solver. In: TACAS, pp. 337–340 (2008)
22. Owre, S., Shankar, N., Rushby, J.M., Stringer-Calvert, D.W.J.: PVS Language Reference, Version 2.4. SRI International (2001)
23. Paulson, L.: Isabelle: A Generic Theorem Prover, vol. 828 of Lecture Notes in Computer Science. Springer (1994)

24. Cok, D.R., Kiniry, J.R.: ESC/Java2: uniting ESC/Java and JML: progress and issues in building and using ESC/Java2, including a case study involving the use of the tool to verify portions of an Internet voting tally system. In: Barthe, G., Burdy, L.,Huisman, M., Lanet, J.-L., Muntean, T. (eds.) Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS 2004), vol. 3362 of Lecture Notes in Computer Science, pp. 108–128. Springer (2005)
25. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software: The KeY Approach. LNCS 4334. Springer (2007)
26. KeY web site. http://www.key-project.org
27. Barnett, M., Leino, K.R.M.: Weakest-precondition of unstructured programs. In: Ernst, M.D., Jensen, T.P. (eds.) Program Analysis for Software Tools and Engineering (PASTE). ACM (September 2005)
28. Leino, K.R.M., Nelson, G., Saxe, J.B.: ESC/Java User's Manual. Technical Note. Compaq Systems Research Center (October 2000)
29. CodeSonar® web site. http://www.grammatech.com/products/codesonar
30. Burdy, L., Requet, A., Lanet, J.-L.: Java Applet Correctness: A Developer-oriented Approach. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003: Formal Methods: International Symposium of Formal Methods Europe, vol. 2805 of Lecture Notes in Computer Science, pp. 422–439. Springer (2003)
31. Cook, J.J.: Reverse Execution of Java Bytecode. Comput. J. **45**, 2002 (2002)
32. Lewis, B.: Debugging Backwards in Time (2003)
33. Ranise, S., Tinelli, C.: The SMT-LIB Standard: Version 1.2. Technical Report. Department of Computer Science, The University of Iowa (2006). Available at http://www.SMT-LIB.org
34. OpenJDK web site. http://openjdk.java.net
35. Beatty, D.L., Bryant, R.E.: Formally verifying a microprocessor using a simulation methodology. In: DAC '94: Proceedings of the 31st Annual Conference on Design Automation, ACM, pp. 596–602. New York, NY, USA (1994)
36. Beer, I., Ben-David, S., Eisner, C., Rodeh, Y.: Efficient detection of vacuity in ACTL formulas. In: CAV '97: Proceedings of the 9th International Conference on Computer Aided Verification, vol. 1254 of Lecture Notes in Computer Science, pp. 279–290. Springer, London, UK (1997)
37. Janota, M., Grigore, R., Moskal, M.: Reachability analysis for annotated code. In: SAVCBS '07: Proceedings of the 2007 Conference on Specification and Verification of Component-based Systems, ACM, pp. 23–30. New York, NY, USA (2007)
38. Egyed, A.: Instant consistency checking for the UML. In: ICSE '06: Proceedings of the 28th International Conference on Software Engineering, ACM, pp. 381–390. New York, NY, USA (2006)
39. Smaragdakis, Y., Csallner, C., Subramanian, R.: Scalable satisfiability checking and test data generation from modeling diagrams. Autom. Softw. Eng. **16**(1) (2009)
40. Leino, K.R.M., Moskal, M., Schulte, W.: Verification condition splitting. Available at http://research.microsoft.com/apps/pubs/default.aspx?id=77373 (2008)